1. **What is the project's general architectural structure and system design?**

    - A "client" (client folder). This is the entry point, and provides access to all of the main functionality that Subxt provides. We have an `OnlineClient` and an `OfflineClient`, which allows us to expose some functionality that can work without any network connection, and some that requires one. We also have a `LightClient` which is essentially an `OnlineClient` that talks to a light client (`smoldot`) rather than to one specific RPC node.
    - Clients require some configuration to know about certain things that can differ between chains (see the config folder). We have built-in config to work with the polkadot relay chain and default substrate node template. A big thing that config defines is how to encode and decode the signed extensions that form a part of any transaction that's crated ("config/extrinsic_params.rs")
    - An `OnlineClient` relies on a `Backend` implementation (see the backend folder) to actually talk to nodes. The `Backend` is the "low level interface" to a node on which everything else is built. We have two Backends; one using the "old" RPC methods exposed by nodes, and one using a set of new RPC methods we're working to stabilise.
    - Both of our `Backend`s use JSON-RPC to communicate with nodes. The "backend/rpc" folder contains the general RPC client interface that is expected to do this, and then an implementation to make the `jsonrpsee` client (see the `jsonrpsee` crate, external) satisfy this. By default then, we use `jsonrpsee` for our RPC communications.
    - `OnlineClient`s expose various sets of high level APIs for interacting with nodes, such as `.tx()` (tx folder) for submitting transactions, `.blocks()` (blocks folder) for downloading and decoding blocks, `.constants()` (constants folder) for decoding constants, `.storage()` (storage folder) for accessing chain storage values, `.runtime_api()` (runtime_api folder) for interacting with chain runtime APIs. Each of the relevant folders contains the interface and logic for each of these things.
    - Interacting with these high level APIs is typically done with the help of generated code. Subxt exposes the `subxt-macro` crate and its derive macro to generate the relevant code to work with thee interfaces. One can also use the `subxt-codegen` crate directly to programmatically generate this interface, or the `subxt` CLI tool to generate it on the CLI.
    - We can also forego this generation and interact with the APIs using values constructed at runtime. See the "dynamic.rs" file, which exports functions to construct dynamic values for the various high level interfaces.
    - Whether static or runtime values are given to the various interfaces, Subxt makes use of metadata (see the metadata folder) to know how to encode and decode values in the right way for a chain/node to understand them. The `subxt-metadata` crate provides a general wrapper around the sort of metadata we can get back from nodes (and can be constructed from different versions of metadata transparently, for instance).

2. **What are the roles in the product and the use cases and workflow for each role?**

Subxt doesn't have any notion of roles. It's a library that anybody can use, with any permissions being handled by the node/chain being communicated to (and requiring transactions to be signed to provide that they are from a given user).

3. **Which function's or module's attack vector do you have the most concern about?**

Two important high level APIs are the ability to submit transactions (this is how you set data on a node), and the ability to read from storage (getting data from a node). Transactions require signing, and for that we have a `subxt-signer` crate, which uses the same cryptographic primitives as Substrate to sign transactions. It's important that this works correctly, but moreover that private keys can't be easily compromised.

Otherwise, we expect the Substrate node to handle most of the "security", and as a client our responsibility is just sending and receiving valid messages to/from it.

4. **Which parts of the system that you want the auditors to focus on?**

    1. Metadata parsing
    2. Singing transactions

5. **What is the trust setup of the system? Should all centralized roles and components be trusted?**

    Because SubXT is a library for communicating with nodes, it should be trusted.

6. **List all the vulnerabilities that are not accepted or already known to the team.**

    No vulnerabilities have been identified for this project.

7. **Is there any other information or docs the auditor should know?**

    1. https://docs.substrate.io/reference/command-line-tools/subxt/
    2. https://docs.rs/subxt/latest/subxt/